Behavior Analysis of Malware by Rewriting-based Abstraction – Extended Version –

Philippe Beaucamps, Isabelle Gnaedig, Jean-Yves Marion
INPL - INRIA Nancy Grand Est - Nancy-Université - LORIA
Campus Scientifique - BP 239 F54506 Vandoeuvre-lès-Nancy Cedex, France
Email: {Philippe.Beaucamps, Isabelle.Gnaedig, Jean-Yves.Marion}@loria.fr

***Abstract*—We propose a formal approach for the detection of high-level program behaviors. These behaviors, defined as combinations of patterns in a signature, are detected by model-checking on abstracted forms of program traces. Our approach works on unbounded sets of traces, which makes our technique useful not only for dynamic analysis, considering one trace at a time, but also for static analysis, considering a set of traces inferred from a control flow graph. Our technique uses a rewriting-based abstraction mechanism, producing a high-level representation of the program behavior, independent of the program implementation. It allows us to handle similar behaviors in a generic way and thus to be robust with respect to variants. Successfully applied to malware detection, our approach allows us in particular to model and detect information leak.**

***Keywords*-behavioral analysis, malware detection, behavior abstraction, model checking, temporal logic, term rewriting, static analysis, information leak**

I. Introduction

Behavior analysis was introduced by Cohen's seminal work [1] to detect malware and in particular unknown malware. In general, a behavior is described by a sequence of system calls and recognition is based on finite state automata [2], [3], [4]. New approaches have been proposed recently. In [5], [6], [7], malicious behaviors are specified by temporal logic formulas with parameters and detection is carried out by model-checking. However, these approaches are tightly dependent on the way malicious actions are realized: using any other system facility to realize an action allows a malware to go undetected. This has motivated yet another approach where a malicious behavior is specified as a combination of high-level actions, in order to be independent from the way these actions are realized and to only consider their effect on a system. In [8] and in [9], a captured execution trace is transformed into a higher-level representation capturing its semantic meaning, i.e., the trace is first abstracted before being compared to a malicious behavior. In [10], the authors propose to use attribute automata, at the price of an exponential time complexity detection. These dynamic abstraction-based approaches, though they can detect unknown viruses whose execution traces exhibit known malicious behaviors, only deal with a single execution trace.

In this paper, we propose a formal approach for high-level behavior analysis. Underpinned by language theory, term rewriting and first order temporal logic, it allows us to determine whether a program exhibits a high-level behavior, expressed by a first order temporal logic formula. Detection is achieved in two steps. First, traces of the program are abstracted in order to reveal the sequences of high-level functionalities they realize. Then, abstracted traces are compared with the behavior formula, using usual model-checking techniques. Functionalities have parameters representing the manipulated data, so our formalism is adapted to the protection against generic threats like the leak of sensitive information.

Our approach has two main characteristics. First, in order to consider a more complete representation of the program than with a single trace, we work on an unbounded set of traces representing its behavior. To deal with the infinity of the set of traces, we restrict to regular sets and safely approximate the set of abstract traces, so that we detect in linear time whether a program exhibits a given behavior. In practice, we represent a program set of traces by a finite state automaton constructed from the program control flow graph.

Second, in order to only keep the essence of the functions performed by the program, to be independent of their possible implementations and to be generic with respect to behavior mutations, we work on abstract forms of traces. Behavior components are abstracted in program traces, by identifying known functionalities and marking them by inserting abstract functionality symbols. These functionalities are described by behavior patterns, which are formally defined, like high-level behaviors, by first-order temporal logic formulas. Thus, the abstract form of an execution trace is defined in terms of these abstract functionalities and not anymore in terms of observed actions, which are low-level and therefore less reliable.

By working on sets of traces, that may consist of a single trace as well as of an unbounded number of traces, our approach may be used for static behavior analysis, i.e., behavior analysis in a static analysis setting.

Static behavior analysis, when it is possible, has many advantages and applications. First, it allows us to analyze the behavior of a program in a more exhaustive way, as it analyzes the unbounded set of the program execution traces, or an approximation of it. Second, static behavior analysis can complement classical, dynamic, behavior analysis with an analysis of the future behavior, when some critical point is

reached in an execution, e.g., when data is about to be sent on the network after some sensitive file was read. Indeed, since dynamic behavior analysis only analyzes the past behavior, the captured data in this case may not be sufficient to raise an alarm, thereby letting the program execute the critical operation.

An interesting application of static behavior analysis is the audit of programs in high-level technologies, like mobile applications, browser extensions, web page scripts, .NET or Java programs. Auditing these programs is complex and mostly manual, resulting in highly publicized infections [11], [12]. In this context, static analysis can provide an appropriate help, because it is usually easier than for general programs, especially when additionally enforcing a security policy (e.g. prohibiting self-modification [13]) or when enforcing strict development guidelines (e.g. for iPhone applications).

Note that the construction of an exhaustive representation of a program behavior is an intractable problem in general: in particular, a program flow may not be easily followed due to indirect jumps, and a program may use complex code protection, for instance by dynamically modifying its code or by using obfuscation. Self modification is usually tackled by emulating the program long enough to deactivate most code protections. Indirect jumps and obfuscation are usually handled by abstract interpretation [14], [15] or symbolic execution [16].

To our knowledge, the use of behavior abstraction on top of static behavior analysis has not been investigated so far.

As our detection mechanism relies on the validation of temporal logic formulas, it is akin to model checking [17], for which there already exists numerous frameworks and tools [18], [19], [20]. The specificity of our approach, however, is that, rather than being applied on the set of program traces, verification is applied on the set of abstract forms of these traces, which is not computable in general. Accordingly, we identify a property of practical high-level behaviors allowing us to approximate this set, in a sound and complete way with respect to detection, and then to apply classical verification techniques.

Our abstraction framework can be used in two scenarios:

- *Detection of given behaviors*: signatures of given high-level behaviors are expressed in terms of abstract functionalities. Given some program, we then assess whether one of its execution traces exhibits a sequence of known functionalities, in a way specific to one of the given behaviors. This can be applied to detection of suspicious behaviors. Although detection of such suspicious behaviors may not suffice to label a program as malicious, it can be used to supplement existing detection techniques with additional decision criteria.
- *Analysis of programs*: abstraction provides a simple and high-level representation of a program behavior, which is more suitable than the original traces for manual analysis, or for analysis of behavior similarity with known malware, etc. For instance, it could be used to detect non necessarily harmful behaviors, in order to get a basic understanding of the program and to further investigate if deemed necessary. It could also be used to automatically discover sequences of high-level functionalities and their dataflow dependencies, exhibited by a program.

Previous work: In [21], we already proposed to abstract program sets of traces with respect to behavior patterns, for detection and analysis. But patterns were defined by string rewriting systems, which did not allow the actions composing a trace to have parameters, precluding dataflow analysis. Moreover abstraction rules replaced identified patterns by abstraction symbols in the original trace, precluding a further detection of patterns interleaved with the rewritten ones.

The formalism proposed in this paper addresses both issues: we handle interleaved patterns by keeping the identified patterns when abstracting them and we express data constraints on action parameters by using term rewriting systems. Finally, another main difference with [21] is that, using the dataflow, we can now detect information leaks in order to prevent unauthorized disclosure or modifications of information.

II. Background

Term Algebras: Let $S = \{Trace, Action, Data\}$ be a set of sorts and $\mathcal{F} = \mathcal{F}_t \cup \mathcal{F}_a \cup \mathcal{F}_d$ be an S-sorted signature, where $\mathcal{F}_t$, $\mathcal{F}_a$, $\mathcal{F}_d$ are mutually distinct and:

- $\mathcal{F}_t = \{\epsilon, \cdot\}$ is the set of the trace constructors;
- $\mathcal{F}_a$ is a finite set of function symbols or constants, with signature $Data^n \rightarrow Action$, $n \in \mathbb{N}$, describing actions;
- $\mathcal{F}_d$ is a finite set of constants of type $Data$, describing data.

We denote by $T(\mathcal{F}, X)$ *the set of S-sorted terms* over a set X of S-sorted variables. For any sort $s \in S$, we denote by $T_s(\mathcal{F}, X)$ *the restriction* of $T(\mathcal{F}, X)$ to terms of sort s and by X_s *the subset of variables* of X of sort s.

If $f \in \mathcal{F}$ is a symbol of arity $n \in \mathbb{N}$, we denote by $f(\overline{x})$ a term $f(x_1, \ldots, x_n)$, where x_1, ..., x_n are variables.

Substitutions are defined as usual (see Appendix A1). By convention, we denote by $t\sigma$ or by $\sigma(t)$ the application of a substitution σ to a term $t \in T(\mathcal{F}, X)$ and by $L\sigma$ the application of σ to a set of terms $L \subseteq T(\mathcal{F}, X)$. The *set of ground substitutions* over X is denoted by $Subst_X$.

A term of sort $Action$ is called an action and a term of sort $Trace$ is called a trace. We distinguish the sort $Action$ from the sort $Trace$ but, for a sake of readability, we may denote by a the trace $\cdot(a, \epsilon)$, for some action a. Similarly, we use the $\cdot$ symbol with infix notation and right associativity, and ϵ is understood when the context is unambiguous. For instance, if a, b, c are actions, $a \cdot b \cdot c$ denotes the trace $\cdot(a, \cdot(b, \cdot(c, \epsilon)))$.

We partition $\mathcal{F}_a$ in a set Σ of symbols, denoting concrete program-level actions, and a set Γ of symbols, denoting abstract actions identifying abstracted functionalities. To construct purely concrete (resp. abstract) terms, we use $\mathcal{F}_\Sigma = \mathcal{F} \setminus \Gamma$ (resp. $\mathcal{F}_\Gamma = \mathcal{F} \setminus \Sigma$).

We define in a natural way the *concatenation* $t \cdot t'$ of two traces t and t'. The *projection* $t|_{\Sigma'}$, also denoted $\pi_{\Sigma'}(t)$, of a trace t on an alphabet $\Sigma' \subseteq \mathcal{F}_a$ corresponds to keeping in a trace only actions from Σ' and is naturally extended to sets of traces (see Appendix A1).

***Program Behavior**:* The representation of a program is chosen to be its set of traces. When executing a program, the captured data is represented on the alphabets Σ, denoting the concrete actions, and $\mathcal{F}_d$, describing the data. In this paper, we consider that the captured data is the library calls along with their arguments. Σ therefore represents the finite set of library calls, while constants from $\mathcal{F}_d$ identify the arguments and the return values of these calls. A *program execution trace* then consists of a sequence of library calls and is defined by a term of $T_{Trace}(\mathcal{F}_\Sigma)$. A *program behavior* is defined by the set of its execution traces, that is a possibly infinite subset of $T_{Trace}(\mathcal{F}_\Sigma)$. For instance, the term $fopen(1,2)\cdot fwrite(1,3)$ represents the execution trace of a file open call $fopen(1,2)$ followed by a file write call $fwrite(1,3)$, where $1 \in \mathcal{F}_d$ identifies the file handle returned by $fopen$, $2 \in \mathcal{F}_d$ identifies the file path and $3 \in \mathcal{F}_d$ identifies the written data.

***First-Order LTL (FOLTL) Temporal Logic**:* We consider the First-Order Temporal Logic (FOLTL) defined in [17], without the equality predicate, where atomic predicates are terms and may have variables. More precisely, let X be a finite set of variables of sort $Data$ and $AP = T_{Action}(\mathcal{F}_\Sigma, X)$ be the set of atomic propositions. FOLTL is an extension of the LTL temporal logic (see Appendix A2) such that:

- If φ is an LTL formula, then φ is an FOLTL formula ;
- If φ is an FOLTL formula and $Y \subseteq X$ is a set of variables, then: $\exists Y.\varphi$ and $\forall Y.\varphi$ are FOLTL formulas, where as usual: $\forall Y.\varphi \equiv \neg\exists Y.\neg\varphi$.

Notation $\varphi_1 \odot \varphi_2$ stands for $\varphi_1 \wedge \mathbf{X}(\top \mathbf{U} \varphi_2)$.

We say that an FOLTL formula is *closed* when it has no free variable, i.e., every variable is bound by a quantifier.

Let $Y \subseteq X$ be a set of variables of sort $Data$ and $\sigma \in Subst_Y$ be a ground substitution over Y. The *application of* σ to an FOLTL formula φ is naturally defined by the formula $\varphi\sigma$ where any free variable x in φ which is in Y has been replaced by its value $\sigma(x)$.

As with LTL, a formula is *validated* on infinite sequences of sets of atomic predicates, denoted by $\xi = (\xi_0, \xi_1, \ldots) \in \left(2^{AP}\right)^\omega$. $\xi \models \varphi$ (ξ validates φ) is defined in the same way as for the LTL logic, with the additional rule: $\xi \models \exists Y.\varphi$ iff there exists a substitution $\sigma \in Subst_Y$ such that $\xi \models \varphi\sigma$.

In our context, a formula is validated over traces of $T_{Trace}(\mathcal{F})$ identified with sequences of singleton sets of atomic predicates. A finite trace $t = a_0 \cdots a_n$ is identified with the infinite sequence of sets of atomic predicates $\xi_t = (\{a_0\}, \ldots, \{a_n\}, \{\}, \{\}, \ldots)$, and t validates φ, denoted by $t \models \varphi$, iff $\xi_t \models \varphi$.

***Tree Automata and Tree Transducers**:* Tree automata and tree transducers are defined as usual (see Appendix A3 and [22]). We consider specifically top-down tree automata without ϵ rules and linear nondeleting top-down tree transducers. A tree language is regular iff it is recognized by some tree automaton and a binary relation is rational iff it is realized by some linear nondeleting top-down tree transducer.

III. Behavior Patterns

The problem under study can be formalized in the following way. First, using FOLTL formulas, we define a set of behavior patterns, where each pattern represents a (possibly infinite) set of terms from $T_{Trace}(\mathcal{F}_\Sigma)$. Second, we need to define a terminating abstraction relation R allowing to schematize a trace by abstracting occurrences of the behavior patterns in that trace. Finally, given some program p coming with an infinite set of traces L (static analysis scenario, for instance by using the control flow graph, see our previous work [21] and [23], [24]), we formulate the *detection problem* in the following way: given an abstract behavior M defined by an FOLTL formula φ, does there exist a trace t in $L{\downarrow_R}$ such that $t \models \varphi$, where $L{\downarrow_R}$ is the set of normal forms of traces of L for R? Our goal is then to find an effective and efficient method solving this problem.

A behavior pattern describes a functionality we want to recognize in a program trace, like writing to system files, sending a mail or pinging a remote host. Such a functionality can be realized in different ways, depending on which system calls, library calls or programming languages it uses.

We describe a functionality by an FOLTL formula, such that traces validating this formula are traces carrying out the functionality.

Example 1. *Let us consider the functionality of sending a ping. One way of realizing it consists in calling the socket function with the parameter* `IPPROTO_ICMP` *describing the network protocol and, then, calling the sendto function with the parameter* `ICMP_ECHOREQ` *describing the data to be sent. Between these two calls, the socket should not be freed . This is described by the FOLTL formula:* $\varphi_1 = \exists x, y.\, socket(x, \alpha) \wedge (\neg closesocket(x)\ \mathbf{U}\ sendto(x, \beta, y))$*, where the first parameter of socket is the created socket and the second parameter is the network protocol, the first parameter of sendto is the used socket, the second parameter is the sent data and the third one is the target, the unique parameter of closesocket is the freed socket and constants* α *and* β *in* $\mathcal{F}_d$ *identify the above parameters* `IPPROTO_ICMP` *and* `ICMP_ECHOREQ`*.*

A ping may also be realized using the function $IcmpSendEcho$*, whose parameter represents the ping target. This corresponds to the FOLTL formula:* $\varphi_2 = \exists x.\, IcmpSendEcho(x)$*.*

Hence, the ping functionality may be described by the FOLTL formula: $\varphi_{ping} = \varphi_1 \vee \varphi_2$*.*

We then define a behavior pattern as the set of traces carrying out its functionality, i.e., as the set of traces validating the formula describing the functionality.

Definition 1. *A* behavior pattern *is a set of traces* $B \subseteq T_{Trace}(\mathcal{F}_\Sigma)$ *validating a closed FOLTL formula* φ *on* $AP = T_{Action}(\mathcal{F}_\Sigma, X)$*:* $B = \{t \in T_{Trace}(\mathcal{F}_\Sigma) \mid t \models \varphi\}$.

IV. Detection Problem

As said before, our goal is to be able to detect, in a given set of traces, some predefined behavior composed of combinations of high-level functionalities. For this, we associate to each behavior pattern an abstract symbol λ taken in the alphabet Γ. An abstract behavior is then defined by combinations of abstract symbols associated to behavior patterns, using an FOLTL formula φ on $AP = T_{Action}(\mathcal{F}_\Gamma, X)$ instead of $T_{Action}(\mathcal{F}_\Sigma, X)$.

Definition 2. *An* abstract behavior *is a set of traces* $M \subseteq T_{Trace}(\mathcal{F}_\Gamma)$ *validating a closed FOLTL formula* φ_M *on* $AP = T_{Action}(\mathcal{F}_\Gamma, X)$*:* $M = \{t \in T_{Trace}(\mathcal{F}_\Gamma) \mid t \models \varphi_M\}$.

When M is defined by a formula φ_M, we write: $M := \varphi_M$.

Example 2. *The abstract behavior of sending a ping to a remote host can then be trivially defined by the formula:* $\varphi_M = \exists x.\, \mathbf{F}\, \lambda_{ping}(x)$.

In the following, for the sake of simplicity, the initial **F** operator is implicit in definitions of abstract behaviors.

Now, let L be the set of program traces we want to analyze. To compare these traces to the given abstract behavior, we have to consider the behavior pattern occurrences they may contain, at the abstract level. For this, we define an abstraction relation R, which marks such occurrences in traces by inserting an abstract symbol λ_B when an occurrence of the behavior pattern B is identified.

From now on, if a behavior pattern is defined using an FOLTL formula φ and associated to an abstraction symbol λ, we may describe it by the notation $\lambda := \varphi$.

The abstraction symbol can have parameters corresponding to those used by the behavior pattern. This allows us to express dataflow constraints in a signature. For instance, the abstraction symbol for the ping behavior pattern can take a parameter denoting the ping target. A signature for a denial of service could then be defined, for example, as a sequence of 100 pings with the same target.

Example 3. *The ping behavior pattern in Example 1 is abstracted in traces by inserting the* λ_{ping} *symbol after the send action or after the IcmpSendEcho action. Then, the trace* $socket(1,\alpha) \cdot gethostbyname(2) \cdot sendto(1,\beta,3) \cdot closesocket(1)$ *can be abstracted into the trace* $socket(1,\alpha) \cdot gethostbyname(2) \cdot sendto(1,\beta,3) \cdot \lambda_{ping}(3) \cdot closesocket(1)$.

Thus, abstraction of a trace reveals abstract behavior pattern combinations, which may constitute the abstract behavior to be observed. In Section VI, we formally define the abstraction relation as a terminating reduction relation induced by a term rewriting system.

Then the detection problem can be defined as follows.

Definition 3. *A set of traces* L exhibits an abstract behavior M *defined by a formula* φ_M*, denoted by* $L \Subset M$*, iff:* $\exists t \in L{\downarrow_R}|_\Gamma$, $t \models \varphi_M$.

When L is restricted to a single trace, or to a finite set of traces, like in dynamic analysis, $L{\downarrow_R}$ is computable since R is terminating. Moreover, as FOLTL quantification is performed over variables in the finite domain $\mathcal{F}_d$, FOLTL verification is decidable, so it can also be decided whether L exhibits M.

In the case of an infinite set of traces L, the computation of $L{\downarrow_R}$ often relies on the computation of the set of descendants $R^*(L)$ of L. But $R^*(L)$ is computable only for some classes of rewrite systems [25] and when L is regular. Unfortunately, the rewrite systems which implement the abstraction relations and which are described in Section VI do not belong to any of these classes. Hence, we cannot rely on the construction of $L{\downarrow_R}$ to decide whether L exhibits M.

However, we will see that, for behaviors considered in practice, a partial abstraction of the set of traces is sufficient i.e., computing the set of normal forms is unnecessary. We therefore propose a detection algorithm relying on a safe approximation of the set of abstract traces. This approximation must be chosen carefully. For instance, it cannot consist in computing, for some n, the set $R^{\leq n}(L)$ of descendants of L until the order n, as shown by the following example.

Example 4. *Let* $\lambda_1 := a$, $\lambda_2 := b$, $\lambda_3 := c$ *be three behavior patterns associated to abstraction relations inserting the abstraction symbol after* a, b *and* c *respectively. Let* $M := \lambda_1 \wedge (\neg\lambda_2\, \mathbf{U}\, \lambda_3)$ *be an abstract behavior. Assume there exists a bound* n *such that* $L{\downarrow_R}$ *may be approximated by* $R^{\leq n}(L)$ *in Definition 3. The trace* $t = a^{n-1} \cdot b \cdot c \cdot d$ *is an example of a sane trace. Yet the trace* $t' = (a \cdot \lambda_1)^{n-1} \cdot b \cdot c \cdot \lambda_3 \cdot d$ *is in* $R^{\leq n}(\{t\})$ *and its projection on* Γ *is in* M*, so we would wrongly infer that* t *exhibits* M.

The problem comes from the fact that $R^{\leq n}(L)$ contains contradictory traces compromising detection i.e., traces seemingly exhibiting an abstract behavior though a few additional abstraction steps would make them leave the signature.

Consequently, we want to exclude traces unreliably realizing the abstract behavior in $R^{\leq n}(L)$, while not having to reach normal forms. In fact, we identify a fundamental property we call (m,n)-completeness, verified by abstract behaviors in practice in the field of malware detection. This property states that, for a program to exhibit an abstract behavior, a necessary and sufficient condition is the following: there exists a partially abstracted trace, abstracted in at most m abstraction steps, realizing the behavior and whose descendants until the order n still realize it.

Definition 4. *Let* M *be an abstract behavior defined by a formula* φ_M *and* m *and* n *be positive numbers.* M *has the property of* (m,n)-completeness *iff for any set of traces* $L \subseteq T_{Trace}(\mathcal{F}_\Sigma)$*:*

$$L \Subset M \;\Leftrightarrow\; \exists t' \in R^{\leq m}(L), \forall t'' \in R^{\leq n}(t')\big|_\Gamma, t'' \models \varphi_M.$$

We then show in the next section that, when L is regular, there exists a sound and complete detection procedure for every abstract behavior enjoying this property. Moreover, the time and space complexity of this detection procedure is linear in the size of the representation of L.

The following theorems show that the (m,n)-completeness property is realistic for abstract behaviors considered in practice.

We first prove that simple abstract behaviors describing sequences of abstract actions with no constraints other than dataflow constraints have the property of (m,n)-completeness.

Theorem 1. *Let* Y *be a set of variables of sort Data. Let* $\alpha_1, \ldots, \alpha_m \in T_{Action}(\mathcal{F}_\Gamma, Y)$*. Then the abstract behavior* $M := \exists Y.\, \alpha_1 \odot \alpha_2 \odot \ldots \odot \alpha_m$ *has the property of* $(m,0)$*-completeness.*

Proofs sketches of the theorems can be found in Appendix C.

We now show that more complex abstract behaviors, forbidding specific abstract actions, have this property.

For a behavior pattern λ, let R_λ denote the restriction of the abstraction relation R to abstraction with respect to λ. We say that two behavior patterns λ and λ' are *independent* iff: $R_\lambda \circ R_{\lambda'} = R_{\lambda'} \circ R_\lambda$. Then we get the following result.

Theorem 2. *Let $M := \exists Y.\, \lambda_1(\overline{x_1}) \wedge \neg(\exists Z.\, \lambda_2(\overline{x_2}))\, \mathbf{U}\, \lambda_3(\overline{x_3})$ be an abstract behavior where Y and Z are two disjoint sets of variables of sort Data and where $\lambda_2 \neq \lambda_1$, $\lambda_2 \neq \lambda_3$ and λ_2 is independent from λ_3. Then M has the property of $(2,1)$-completeness.*

In practice, as we will see in Section VII, most signatures are disjunctions of formulas of the form: $\exists Y.\, \alpha_1 \odot \alpha_2 \odot \ldots \odot \alpha_m$, from Theorem 1, or of the form:

$$\begin{aligned}\exists Y. \lambda_1\left(\overline{x_1}\right) \wedge \neg\left(\exists Z_1.\, \lambda\left(\overline{z_1}\right)\right)\ \mathbf{U}\, \lambda_2\left(\overline{x_2}\right) \wedge \\ \neg\left(\exists Z_2.\, \lambda\left(\overline{z_2}\right)\right)\ \mathbf{U}\ \ldots \lambda_k\left(\overline{x_k}\right).\end{aligned}$$

where λ is independent from λ_2, ..., λ_k. From the proof of Theorem 2, we conjecture that the second formula has the property of $(k,1)$-completeness.

The independence condition is not necessary in general, in order to guarantee that such abstract behaviors have a property of (m,n)-completeness for some m and n, but absence of this condition results in significantly higher values of m and n.

Fundamentally, by Definition 3, detection of an abstract behavior is decomposed into two independent steps: an abstraction step followed by a verification step. The first step computes the abstract forms of the program traces while the second step applies usual verification techniques in order to decide whether one of the computed traces verifies the FOLTL formula defining the abstract behavior. However, when using the (m,n)-completeness property to bypass the general intractability of the abstraction step, this relies on computing a set $\left\{t \in T_{Trace}\left(\mathcal{F}\right),\, R^{\leq n}(t) \models \varphi_M\right\}$ and then intersecting it with $R^{\leq m}(L)$. So we lose the previous decomposition, thereby preventing us from leveraging powerful techniques from the model checking theory. We therefore show that, in the previous theorem, (m,n)-completeness allows us to nonetheless preserve that decomposition, so that the abstraction step now becomes decidable.

Theorem 3. *Let M be an abstract behavior defined by a formula $\varphi_M = \exists Y.\, \lambda_1(\overline{x_1}) \wedge \neg(\exists Z.\, \lambda_2(\overline{x_2}))\, \mathbf{U}\, \lambda_3(\overline{x_3})$ where Y and Z are disjoint sets of variables of sort Data and where $\lambda_2 \neq \lambda_1$, $\lambda_2 \neq \lambda_3$ and λ_2 is independent from λ_3. Then, for any set of traces $L \subseteq T_{Trace}\left(\mathcal{F}_\Sigma\right)$, L exhibits M iff:*

$$\exists t \in\ R_{\lambda_2}\!\downarrow\left(R^{\leq 2}(L)\right)\big|_\Gamma,\, t \models \varphi_M.$$

When both the abstraction relation R and the relation $R_{\lambda_2}\!\downarrow$ are rational, the set $R_{\lambda_2}\!\downarrow\left(R^{\leq 2}(L)\right)$ is computable and regular, and detection then boils down to a classical model checking problem. In the general case, $R_{\lambda_2}\!\downarrow$ is not *rational*, but in our experimentations, the behavior pattern λ_2 is defined by sets A_i and B_i where A_i contains traces made of a single action and $B_i = \{\epsilon\}$. Thus constructing a transducer realizing the relation $R_{\lambda_2}\!\downarrow$ is straightforward.

Remark 1. *An equivalent definition of infection could consist in compiling the abstract behavior, that is computing the set $\pi_\Gamma^{-1}(M)\!\downarrow_{R^{-1}}$ of concrete traces exhibiting M. Then a set of traces L would exhibit M iff one of its subtraces is in this set. This definition seems more intuitive: rather than abstracting a trace and comparing it to an abstract behavior, we check whether this trace is an implementation of the behavior. However, this approach would require to first compute the compiled form of the abstract behavior, $\pi_\Gamma^{-1}(M)\!\downarrow_{R^{-1}}$, which is not generally computable and whose representation can quickly have a prohibitive complexity stemming from the interleaving of behavior patterns occurrences (especially when traces realizing the behavior patterns are complex) and from the variables instantiations.*

V. Detection complexity

The detection problem, like the more general problem of program analysis, requires computing a partial abstraction of the set of analyzed traces. In practice, in order to manipulate this set, we consider a regular approximation of it i.e., a tree automaton. Moreover, we will see in Section VI that, in practice, the abstraction relation is rational i.e., it is realized by a tree transducer.

This entails the decidability of detection.

Theorem 4. *Let R be an abstraction relation, such that R and R^{-1} are rational. There exists a detection procedure deciding whether L exhibits M, for any regular set of traces L and for any regular abstract behavior M having the property of (m,n)-completeness for some positive integers m and n.*

Definition 5. *Let M be an abstract behavior having the property of (m,n)-completeness. The* set of traces n-reliably realizing *M w.r.t an abstraction relation R is the set $\left\{t \in T_{Trace}\left(\mathcal{F}\right) \mid \forall t' \in R^{\leq n}\left(t\right)\big|_\Gamma, t' \models \varphi_M\right\}$.*

Using the set of traces n-reliably realizing M, we get the following detection complexity, which is linear in the size of the automaton recognizing the program set of traces, a major improvement on the exponential complexity bound of [10].

Theorem 5. *Let R be an abstraction relation such that R and R^{-1} are rational. Let τ be a tree transducer realizing R. Let M be a regular abstract behavior with the property of (m,n)-completeness and A_M be a tree automaton recognizing the set of traces n-reliably realizing M w.r.t. R. Deciding whether a regular set of traces L, recognized by a tree automaton A, exhibits M takes $O\left(|\tau|^{m\cdot(m+1)/2} \times |A| \times |A_M|\right)$ time and space.*

VI. Trace Abstraction

As said above, abstracting a trace with respect to some behavior pattern amounts to transforming it when it contains an occurrence of the behavior pattern, by inserting a symbol of Γ in the trace. This symbol, called abstraction symbol, is inserted at the position after which the behavior pattern functionality has been performed. This position is the most logical one to stick to the trace semantics. Furthermore, when behavior patterns appear interleaved, this position allows us to define the order in which their functionalities are realized (see Appendix B for an example).

As said in the introduction, rather than replacing behavior pattern occurrences with abstraction symbols, we preserve them in order to properly handle interleaved behavior patterns occurrences.

Now, let us consider the following example.

Example 5. *Abstraction of the ping in Example 3 is realized by rewriting using the rule* $A_1(x,y) \cdot B_1(x,y) \rightarrow A_1(x,y) \cdot \lambda(y) \cdot B_1(x,y)$*, where* $A_1(x,y) = socket(x,\alpha) \cdot (T_{Trace}(\mathcal{F}_\Sigma) \setminus (T_{Trace}(\mathcal{F}_\Sigma) \cdot closesocket(x) \cdot T_{Trace}(\mathcal{F}_\Sigma))) \cdot sendto(x,\beta,y)$ *and* $B_1(x,y) = \{\epsilon\}$*, and the rule* $A_2(x) \cdot B_2(x) \rightarrow A_2(x) \cdot \lambda(x) \cdot B_2(x)$*, where* $A_2(x) = \{IcmpSendEcho(x)\}$ *and* $B_2(x) = \{\epsilon\}$.

So we define the abstraction relation by decomposing the behavior pattern into a finite union of concatenations of sets $A_i(X)$ and $B_i(X)$ such that traces in $A_i(X)$ end with the action effectively performing the behavior pattern functionality. These sets $A_i(X)$ and $B_i(X)$ are composed of concrete traces only, since abstract actions that may appear in a partially rewritten trace should not impact the abstraction of an occurrence of the behavior pattern.

Definition 6. *Let* $\lambda \in \Gamma$ *be an abstraction symbol,* X *be a set of variables of sort* $Data$*,* $\overline{x}$ *be a sequence of variables in* X*. An* abstraction system *on* $T_{Trace}(\mathcal{F}, X)$ *is a finite set of rewrite rules of the form:* $A_i(X) \cdot B_i(X) \rightarrow A_i(X) \cdot \lambda(\overline{x}) \cdot B_i(X)$ *where the sets* $A_i(X)$ *and* $B_i(X)$ *are sets of concrete traces of* $T_{Trace}(\mathcal{F}_\Sigma, X)$.

The system of rewrite rules we use generates a reduction relation on $T_{Trace}(\mathcal{F})$ such that filtering works on traces projected on Σ.

Definition 7. *The* reduction relation *on* $T_{Trace}(\mathcal{F})$ *generated by a system of* n *rewrite rules* $A_i(X) \cdot B_i(X) \rightarrow A_i(X) \cdot \lambda(\overline{x}) \cdot B_i(X)$ *is the rewriting relation* $\rightarrow_{\mathcal{R}}$ *such that, for all* $t, t' \in T_{Trace}(\mathcal{F}), t \rightarrow_{\mathcal{R}} t'$ *iff:*

$$\exists \sigma \in Subst_X, \exists p \in Pos(t), \exists i \in [1..n],$$
$$\exists a \in T_{Trace}(\mathcal{F}) \cdot T_{Action}(\mathcal{F}_\Sigma), \exists b, u \in T_{Trace}(\mathcal{F}),$$
$$a|_\Sigma \in A_i(X)\sigma,\ b|_\Sigma \in B_i(X)\sigma,\ t|_p = a \cdot b \cdot u$$
$$\text{and } t' = t[a \cdot \lambda(\overline{x})\sigma \cdot b \cdot u]_p.$$

An abstraction relation with respect to a given behavior pattern is thus the reduction relation of an abstraction system, where left members of the rules cover the set of the traces realizing the behavior pattern functionality.

Definition 8. *Let* B *be a behavior pattern associated with an abstraction symbol* $\lambda \in \Gamma$*. Let* X *be a set of variables of sort* $Data$*. An* abstraction relation *w.r.t. this behavior pattern is the reduction relation on* $T_{Trace}(\mathcal{F})$ *generated by an abstraction system composed of* n *rules* $A_i(X) \cdot B_i(X) \rightarrow A_i(X) \cdot \lambda(\overline{x}) \cdot B_i(X)$ *verifying:*

$$B = \bigcup_{i \in [1..n]} \bigcup_{\sigma \in Subst_X} (A_i(X) \cdot B_i(X))\sigma.$$

Finally, we generalize the definition of abstraction to a set of behavior patterns.

Definition 9. *Let* C *be a finite set of behavior patterns. An* abstraction relation *w.r.t* C *is the union of the abstraction relations w.r.t. each behavior pattern of* C.

Total Abstraction

If R is an abstraction relation with respect to our set of behavior patterns, we want to define the total abstraction $L{\downarrow_R}$.

Even in the case of a finite set of traces L, abstraction does not terminate in general, since the same occurrence of a pattern can be abstracted an unbounded number of times. We therefore require that the same abstract action is not inserted twice after the same concrete action. In other words, if a term $t = t_1 \cdot t_2$ is abstracted into a term $t' = t_1 \cdot \alpha \cdot t_2$, where α is the inserted abstract action, then if t_2 starts with a sequence of abstract actions, α does not appear in this sequence.

Definition 10. *The* terminating abstraction relation *for an abstraction relation* R *is the relation* R' *defined by:* $\forall t_1, t_2 \in T_{Trace}(\mathcal{F}), \forall \alpha \in T_{Action}(\mathcal{F}_\Gamma), t_1 \cdot t_2 \rightarrow_{R'} t_1 \cdot \alpha \cdot t_2 \Leftrightarrow t_1 \cdot t_2 \rightarrow_R t_1 \cdot \alpha \cdot t_2$ *and* $\nexists u \in T_{Trace}(\mathcal{F}_\Gamma), \nexists u' \in T_{Trace}(\mathcal{F}), t_2 = u \cdot \alpha \cdot u'$.

Using the above definition, a behavior pattern occurrence can only be abstracted once. Furthermore, abstraction does not create new abstraction opportunities so the relation R' is clearly terminating.

Remark 2. *Note that a terminating abstraction relation with respect to a set of behavior patterns is not confluent in general. We could adapt the definition of the abstraction relation to make it confluent, for instance by defining an order on the set* $T_{Action}(\mathcal{F}_\Gamma)$*. However, as we have seen in Section IV, detection works on the set of normal forms. So having several normal forms for a trace does not compromise its mechanism.*

Rational Abstraction

In practice, a behavior pattern is regular, along with the set of instances of right-hand sides of its abstraction rules. We show that this is sufficient to ensure that the abstraction relation is realizable by a tree transducer, in other words that it is a rational tree transduction. The tree transducer formalism is chosen for its interesting formal (closure by union, composition, preservation of regularity) and computational properties.

Theorem 6. *Let* B *be a behavior pattern and* R *be a terminating abstraction relation w.r.t.* B *defined by an abstraction system whose set of instances of right-hand sides of rules is recognized by a tree automaton* A_R*. Then* R *and* R^{-1} *are rational and, for any tree automaton* A*,* $R(\mathcal{L}(A))$ *is recognized by a tree automaton of size* $O(|A| \cdot |A_R|)$

VII. Application to Information Leak Detection

Abstraction can be applied to detection of generic threats, and in particular to detection of sensitive information leak. Such a leak can be decomposed into two steps: capturing sensitive information and sending this information to an exogenous location. The captured data can be keystrokes, passwords or data read from a sensitive network location, while the

exogenous location can be the network, a removable device, etc. Thus, we define a behavior pattern $\lambda_{steal}(x)$, representing the capture of some sensitive data x, and a behavior pattern $\lambda_{leak}(x)$, representing the transmission of x to an exogenous location. Moreover, since the captured data must not be invalidated before being leaked, we define a behavior pattern $\lambda_{inval}(x)$, which represents such an invalidation. The information leak abstract behavior is then defined by:

$$M := \exists x.\, \lambda_{steal}(x) \wedge \neg\lambda_{inval}(x)\ \mathbf{U}\ \lambda_{leak}(x).$$

By looking at several malware samples, like keyloggers, sms message leaking applications or personal information stealing mobile applications, we consider the following definitions of the three behavior patterns involved:

- $\lambda_{steal}(x)$ describes a keystroke capture functionality[1] and, on Android mobile phones, the retrieving of the IMEI number:

$$\begin{array}{c}\lambda_{steal}(x) := GetAsyncKeyState(x) \vee \\ (RegisterDev(\texttt{KBD}, \texttt{SINK}) \odot GetInputData(x, \texttt{INPUT})) \vee \\ (\exists y.\, SetWindowsHookEx(y, \texttt{WH\_KEYBOARD\_LL}) \wedge \\ \neg UnhookWindowsHookEx(y)\mathbf{U}\ HookCalled(y, x)) \vee \\ \exists y. TelephonyManager\_getDeviceId(x, y).\end{array}$$

- $\lambda_{leak}(x)$ describes a network send functionality under Windows or Android:

$$\begin{array}{c}\lambda_{leak}(x) := \exists y, z.\, sendto(z, x, y) \vee \\ \exists y, z.\, (connect(z, y) \wedge \neg close(z)\ \mathbf{U}\ send(z, x)) \vee \\ \exists c, s.\, HttpURLConnection\_getOutputStream(s, c) \wedge \\ \neg OutputStream\_close(s)\mathbf{U}\ OutputStream\_write(s, x).\end{array}$$

- $\lambda_{inval}(x)$ describes the overwriting or freeing of x:

$$\begin{array}{c}\lambda_{inval}(x) := free(x) \vee \exists y.\, sprintf_0(x, y) \vee \\ GetInputData(x, \texttt{INPUT}) \vee \ldots\end{array}$$

Finally, the captured data is usually not leaked in its raw form, so we take into account transformations of this data via the behavior pattern $\lambda_{depends}(x, y)$ which denotes a dependency of x on y. For instance, x may be a string representation of y, or x may be an encryption or an encoding of y:

$$\begin{array}{l}\lambda_{depends}(x, y) := sprintf_0(x, y) \vee \exists s.\, sprintf_1(x, s, y) \vee \\ \exists sb.\, StringBuilder\_append(sb, y) \odot SB\_toString(x, sb).\end{array}$$

Then, in order to account for one such transformation of the stolen data, we adapt the definition of the information leak abstract behavior:

$$\begin{array}{l}M := \exists x, y.\, \lambda_{steal}(x) \wedge \neg\lambda_{inval}(x)\ \mathbf{U}\ \lambda_{depends}(y, x) \wedge \\ \qquad\qquad \neg\lambda_{inval}(y)\ \mathbf{U}\ \lambda_{leak}(y).\end{array}$$

Of course, we can adapt this formula to allow more than one data transformation.

[1]We assume the execution of a hook f with argument x is represented in a trace by an action $HookCalled(f, x)$.

VIII. Experiments

We tested the validity of our approach on several types of malware: keyloggers, sms message leaking, mobile phone personal information stealing. Our goal is to detect the information leak behavior M defined in the previous section. In order to perform behavior pattern abstraction and behavior detection in the presence of data, we use the CADP toolbox [26], which allows us to manipulate and model-check communicating processes written in the LOTOS language. CADP features a verification tool, evaluator4, which allows on-the-fly model checking of formulas expressed in the MCL language, a fragment of the modal mu-calculus extended with data variables, whose FOLTL logic used in this paper is a subset.

We first represent the program set of traces as a CADP process. For this, we use a program control flow graph obtained by static analysis (see [21] and [23], [24]). Regularity of the set of traces is enforced by limiting recursion and inlining function calls, an approximation that can be deemed safe with respect to the abstract behaviors to detect. Note that there are two shortcomings to regular approximation. First, approximation of conditional branches by nondeterministic branches may result in false positives, especially when the program code is obfuscated. And second, failure to identify data correlations during dataflow analysis can result in false negatives. However, this does not significantly impact our detection results.

Now, as expressed in Theorem 3, detection of the information leak abstract behavior M can be broken down into two steps: abstracting the set of traces L by computing $R_{\lambda_{inval}}\downarrow\left(R^{\leq 2}(L)\right)$ and then verifying whether an abstracted trace matches the abstract behavior formula.

So, we can simulate the abstraction step in CADP and delegate the verification step to the evaluator4 module. For this, we represent the set of traces L of a given program by a system of communicating processes expressed in LOTOS, with a particular gate on which communications correspond to library calls. Then, computation of $R^{\leq 2}(L)$ is performed by synchronization with another LOTOS process which simulates the transducer realizing the abstraction. Moreover, the relation $R_{\lambda_{inval}}\downarrow$ is rational and can also be simulated by process synchronization in CADP.

For each malware sample we tested, we successfully run evaluator4 on the resulting process, representing $R_{\lambda_{inval}}\downarrow(R^{\leq 2}(L))$, in order to detect the information leak abstract behavior defined in the previous section.

Also, in our previous work [21], we implemented our string rewriting based abstraction technique and we defined several behavior patterns and abstract behaviors, by looking at malicious execution traces. Then, we tested it on samples of malicious programs collected using a honeypot[2] and identified using Kaspersky Antivirus. These samples belonged to known malware families, among which Allaple, Virut, Agent, Rbot, Afcore and Mimail. Most of them were successfully matched to our malware database.

[2]The honeypot of the Loria's High Security Lab: http://lhs.loria.fr.

IX. Conclusion

We presented an original approach for detecting high-level behaviors in programs, describing combinations of functionalities and defined by first-order temporal logic formulas. Behavior patterns, expressing concrete realizations of the functionalities, are also defined by first-order temporal logic formulas. Abstraction of these functionalities in program traces is performed by term rewriting. Validation of the abstracted traces with respect to some high-level behavior is performed using usual model checking techniques. In order to address the general intractability of the problem of constructing the set of normal forms of traces for a given program, we have identified a property of practical high-level behaviors allowing us to avoid computing normal forms and yielding a linear time detection algorithm.

Abstraction is a key notion of our approach. Providing an abstracted form for program traces and behaviors allows us to be independent of the program implementation and to handle similar behaviors in a generic way, making this framework robust with respect to variants. The fact that high-level behaviors are combinations of elementary patterns enables to efficiently summarize and compact the possible combinations likely to compose suspicious behaviors. Moreover, high-level behaviors and behavior patterns are easy to update since they are expressed in terms of basic blocks.

Our technique could be improved to account for shortcomings in the computation of the dataflow. Indeed, as dataflow is computed by static analysis, some relations between data may have been wrongly inferred or ignored. So, we could extend our formalism by considering probabilistic abstraction: when an occurrence of a behavior pattern cannot be precisely matched due to dataflow inconsistencies, we could nevertheless abstract the occurrence by taking the related uncertainty into account. Then, a high-level behavior would be matched on the condition that some abstracted trace realizes it with a high enough certainty. We are currently extending our formalism to include such scenarios.

References

[1] F. Cohen, "Computer viruses: Theory and experiments," *Computers and Security*, vol. 6, no. 1, pp. 22–35, 1987.

[2] B. Le Charlier, A. Mounji, and M. Swimmer, "Dynamic detection and classification of computer viruses using general behaviour patterns," in *International Virus Bulletin Conference*, 1995, pp. 1–22.

[3] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2001, pp. 144–155.

[4] J. Morales, P. Clarke, Y. Deng, and G. Kibria, "Characterization of virus replication," *Journal in Computer Virology*, vol. 4, no. 3, pp. 221–234, August 2007.

[5] J. Bergeron, M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie, and N. Tawbi, "Static detection of malicious code in executable programs," in *Symposium on Requirements Engineering for Information Security*, 2001.

[6] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Detecting malicious code by model checking," in *Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, ser. Lecture Notes in Computer Science, vol. 3548. Springer, 2005, pp. 174–187.

[7] P. K. Singh and A. Lakhotia, "Static verification of worm and virus behavior in binary executables using model checking," in *Information Assurance Workshop*. IEEE Press, 2003, pp. 298–300.

[8] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell, "A layered architecture for detecting malicious behaviors," in *International symposium on Recent Advances in Intrusion Detection*, ser. Lecture Notes in Computer Science, vol. 5230. Springer, 2008, pp. 78–97.

[9] U. Bayer, P. Milani Comparetti, C. Hlauscheck, C. Kruegel, and E. Kirda, "Scalable, Behavior-Based Malware Clustering," in *16th Symposium on Network and Distributed System Security (NDSS)*, 2009.

[10] G. Jacob, H. Debar, and E. Filiol, "Malware behavioral detection by attribute-automata using abstraction from platform and language," in *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, ser. RAID '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 81–100.

[11] "Security Issue on AMO," http://blog.mozilla.com/addons/ 2010/02/04/ please-read-security-issue-on-amo.

[12] "Aftermath of the Droid Dream Android Market Malware Attack," http://nakedsecurity.sophos.com/2011/03/03/ droid-dream-android-market-malware-attack-aftermath/.

[13] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A sandbox for portable, untrusted x86 native code," *Communications of the ACM*, vol. 53, no. 1, pp. 91–99, 2010.

[14] M. D. Preda, M. Christodorescu, S. Jha, and S. Debray, "A semantics-based approach to malware detection," in *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 2007, pp. 377–388.

[15] J. Kinder, H. Veith, and F. Zuleger, "An abstract interpretation-based framework for control flow reconstruction from binaries," in *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2009)*, ser. LNCS, M. M.-O. Neil D. Jones, Ed., vol. 5403. Springer, 2009, pp. 214–228.

[16] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and D. Song, "BitScope: Automatically dissecting malicious binaries," School of Computer Science, Carnegie Mellon University, Tech. Rep. CS-07-133, Mar. 2007.

[17] F. Kröger and S. Merz, *Temporal Logic and State Systems*, ser. Texts in Theoretical Computer Science. An EATCS Series, 2008.

[18] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, sep 2003.

[19] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes," in *Computer Aided Verification (CAV'2007)*, ser. Lecture Notes in Computer Science, Werner Damm and Holger Hermanns, Eds., vol. 4590, Berlin Germany, 2007, pp. 158–163.

[20] F. Chen and G. Roşu, "MOP: An Efficient and Generic Runtime Verification Framework," in *Object-Oriented Programming, Systems, Languages and ApplicationsObject-Oriented Programming, Systems, Languages and Applications*. ACM press, 2007, pp. 569–588.

[21] P. Beaucamps, I. Gnaedig, and J.-Y. Marion, "Behavior Abstraction in Malware Analysis," in *1st International Conference on Runtime Verification*, ser. Lecture Notes in Computer Science, O. S. Grigore Rosu, Ed., vol. 6418. St. Julians Malta: Springer-Verlag, Aug. 2010, pp. 168–182. [Online]. Available: http://hal.inria.fr/inria-00536500/en/

[22] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi, "Tree automata techniques and applications," Available on: http://www.grappa.univ-lille3.fr/tata, 2007.

[23] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2005, pp. 32–46.

[24] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer, "Behavior-based Spyware Detection," in *Proceedings of the 15th USENIX Security Symposium*, Vancouver, BC, Canada, August 2006.

[25] R. Gilleron and S. Tison, "Regular Tree Languages and Rewrite Systems," *Fundamenta Informaticae*, vol. 24, pp. 157–176, 1995. [Online]. Available: http://hal.inria.fr/inria-00538882/en/

[26] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "Cadp 2010: A toolbox for the construction and analysis of distributed processes," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, P. Abdulla and K. Leino, Eds. Springer Berlin / Heidelberg, 2011, vol. 6605, pp. 372–387.

[27] M. Mohri, "Statistical natural language processing," in *Applied Combinatorics on Words*. New York, NY, USA: Cambridge University Press, 2005, ch. 4.

APPENDIX

A. Additional Background

1) Term Algebras: A ground substitution on a finite set X of S-sorted variables is a mapping $\sigma : X \to T(\mathcal{F})$ such that: $\forall s \in S, \forall x \in X_s,\ \sigma(x) \in T_s(\mathcal{F})$. σ can be naturally extended to a mapping $T(\mathcal{F}, X) \to T(\mathcal{F})$ in such a way that:

$$\begin{gathered}\forall f(t_1, \ldots, t_n) \in T(\mathcal{F}, X),\\ \sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n)).\end{gathered}$$

If X is a set of variables of sort $Data$, we define the projection on an alphabet $\Sigma' \subseteq \mathcal{F}_a$ of a term $t \in T_{Trace}(\mathcal{F}, X)$, denoted by $\pi_{\Sigma'}(t)$ or, equivalently, by $t|_{\Sigma'}$, in the following way:

$$\begin{aligned}\pi_{\Sigma'}(\epsilon) &= \epsilon\\ \pi_{\Sigma'}(b \cdot u) &= \begin{cases} b \cdot \pi_{\Sigma'}(u) & \text{if } b \in T_{Action}(\mathcal{F}_{\Sigma'}, X)\\ \pi_{\Sigma'}(u) & \text{otherwise}\end{cases}\end{aligned}$$

with $b \in T_{Action}(\mathcal{F}, X)$ and $u \in T_{Trace}(\mathcal{F}, X)$.

Similarly, the concatenation of two terms t and t' of $T_{Trace}(\mathcal{F}, X)$, where X is a set of S-sorted variables and $t \notin X$, is denoted by $t \cdot t' \in T_{Trace}(\mathcal{F}, X)$ and defined by $t \cdot t' = t[t']_p$, where p is the position of ϵ in t, i.e., $t|_p = \epsilon$. Projection and concatenation are naturally extended to sets of terms of sort $Trace$. We also extend concatenation to $2^{T_{Trace}(\mathcal{F},X)} \times 2^{T_{Trace}(\mathcal{F},X)}$ with $L \cdot L' = \{t \cdot t' \mid t \in L, t' \in L'\}$ and to $2^{T_{Trace}(\mathcal{F},X)} \times T_{Action}(\mathcal{F}, X)$ with $L \cdot a = L \cdot \{a \cdot \epsilon\}$.

2) LTL Temporal Logic: Let A be an alphabet. We denote by A^ω the set of infinite words over A: $A^\omega = \{a_1 a_2 \ldots \mid \forall i,\ a_i \in A\}$.

Let AP be the set of atomic propositions. An LTL formula is defined as follows:

- $\top$ (true) and $\bot$ (false) are LTL formulas;
- If $p \in AP$, then p is an LTL formula;
- If φ_1 and φ_2 are LTL formulas, then: $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\mathbf{X}\varphi_1$ ("next time"), $\mathbf{F}\varphi_1$ ("eventually" or "in the future") and $\varphi_1 \,\mathbf{U}\, \varphi_2$ ("until") are LTL formulas.

A formula is satisfied on infinite sequences of sets of atomic predicates, denoted by $\xi = (\xi_0, \xi_1, \ldots) \in \left(2^{AP}\right)^\omega$. We denote by ξ^i the sequence $(\xi_i, \xi_{i+1}, \ldots)$. $\xi \models \varphi$ (ξ validates φ) is defined by:

- $\xi \models \top$;
- $\xi \models p$, where $p \in AP$, iff $p \in \xi_0$;
- $\xi \models \neg\varphi$ iff $\xi \not\models \varphi$;
- $\xi \models \varphi_1 \wedge \varphi_2$ iff $\xi \models \varphi_1$ and $\xi \models \varphi_2$;
- $\xi \models \varphi_1 \vee \varphi_2$ iff $\xi \models \varphi_1$ or $\xi \models \varphi_2$;
- $\xi \models \mathbf{X}\varphi$ iff $\xi^1 \models \varphi$;
- $\xi \models \mathbf{F}\varphi$ iff for some $i \geq 0$, $\xi^i \models \varphi$;
- $\xi \models \varphi_1 \mathbf{U} \varphi_2$ iff for some $i \geq 0$, $\xi^i \models \varphi_2$ and, for any $j \in [0..i-1]$, $\xi^j \models \varphi_1$;

3) Tree Automata: Let X be a set of variables. A (top-down) tree automaton [22] is a tuple $A = (\mathcal{F}, Q, q_0, \Delta)$ where $\mathcal{F}$ is a finite alphabet, Q is a finite set of states, $q_0 \in Q$ is an initial state and Δ is a set of rules of the form:

$$q(f(x_1, \ldots, x_n)) \to f(q_1(x_1), \ldots, q_n(x_n))$$

where $f \in \mathcal{F}$ of arity $n \in \mathbb{N}$, $q, q_1, \ldots, q_n \in Q$ and $x_1, \ldots, x_n \in X$.

The transition relation $\to_A$ associated with the automaton A is defined by:

$$\begin{gathered}\forall t, t' \in T(\mathcal{F} \cup Q),\\ t \to_A t'\\ \Leftrightarrow\\ \exists\, q(f(x_1, \ldots, x_n)) \to f(q_1(x_1), \ldots, q_n(x_n)) \in \Delta,\\ \exists p \in Pos(t), \exists u_1, \ldots, u_n \in T(\mathcal{F}),\\ t|_p = q(f(u_1, \ldots, u_n))\\ \text{and } t' = t[f(q_1(u_1), \ldots, q_n(u_n))]_p.\end{gathered}$$

The language recognized by A is defined by: $\mathcal{L}(A) = \{t \mid q_0(t) \to_A^* t\}$. The tree languages recognized by (top-down) tree automata are the regular tree languages.

The size of A is defined by: $|A| = |Q| + |\Delta|$.

4) Tree Transducers: Let X be a set of variables. A (top-down) tree transducer [22] is a tuple $\tau = (\mathcal{F}, \mathcal{F}', Q, q_0, \Delta)$ where $\mathcal{F}$ is the finite set of input symbols, $\mathcal{F}'$ is the finite set of output symbols, Q is a finite set of unary states, $q_0 \in Q$ is the initial state, Δ is a set of transduction rules of the form:

$$\begin{gathered}q(f(x_1, \ldots, x_n)) \xrightarrow{w} u\\ \text{or}\\ q(x_1) \xrightarrow{w} u \qquad\qquad (\epsilon\text{-rule})\end{gathered}$$

where $q \in Q$, $n \in \mathbb{N}$, $f \in \mathcal{F}$ of arity $n \in \mathbb{N}$, $x_1, \ldots, x_n$ are distinct variables from X, $u \in T(\mathcal{F}' \cup Q, \{x_1, \ldots, x_n\})$ and $w \in S$.

The transition relation $\to_\tau$ associated with the transducer τ is defined by:

$$\begin{gathered}\forall t, t' \in T(\mathcal{F} \cup \mathcal{F}' \cup Q),\\ t \to_\tau t'\\ \Leftrightarrow\\ \exists\, q(f(x_1, \ldots, x_n)) \to u \in \Delta,\\ \exists p \in Pos(t), \exists u_1, \ldots, u_n \in T(\mathcal{F}'),\\ t|_p = q(f(u_1, \ldots, u_n))\\ \text{and } t' = t[u\{x_1 \leftarrow u_1, \ldots, x_n \leftarrow u_n\}]_p.\end{gathered}$$

ϵ-rules are a particular case of this definition.

The transduction relation induced by τ is the relation R_τ defined by: $R_\tau = \{(t, t') \mid q_0(t) \to_\tau^* t', t \in T(\mathcal{F}), t' \in T(\mathcal{F}')\}$. A top-down tree transducer is linear (resp. nondeleting) iff its rules are linear (resp. nondeleting). A binary relation in $T(\mathcal{F}, X) \times T(\mathcal{F}', X)$ is called rational iff there exists a linear nondeleting top-down tree transducer realizing it.

The size of τ is defined by: $|\tau| = |Q| + |\Delta|$.

The image of a regular set in $T_{Trace}(\mathcal{F}, X)$ by a rational transduction is a regular set in $T_{Trace}(\mathcal{F}, X)$. Rational tree transductions are closed by union and functional composition. In this paper, we only consider linear nondeleting top-down tree transducers.

In the following lemma, we call trace language on an alphabet $\mathcal{F}$ a tree language in $T_{Trace}(\mathcal{F})$, trace automaton on $\mathcal{F}$ a tree automaton recognizing a trace language and trace transducer a transducer transforming trace languages into trace languages.

Lemma 1. *Let A be a trace automaton on an alphabet $\mathcal{F}$ and $\tau = (\mathcal{F}, \mathcal{F}', Q, q_0, \Delta)$ be a trace transducer. Then the trace*

language $\tau(\mathcal{L}(A))$ is recognized by a trace automaton of size $O(|A| \cdot |\tau|)$.

Proof: Define two word alphabets Ω and Ω' in bijection with the finite sets $T_{Action}(\mathcal{F})$ and $T_{Action}(\mathcal{F}')$.

Then, since A is a trace automaton and τ is a trace transducer, the result follows by direct analogy with the case of string transducers on the word alphabets Ω and Ω' [27]. ■

B. Abstraction examples

Although Example 5 defines a single abstraction rule for each way of realizing the behavior pattern functionality, a realization may sometimes be associated to more than one abstraction rule, especially when constraints of the behavior pattern are complex. This is shown by the following example.

Example 6. *Let B be a behavior pattern constructed from a trace $a(x) \cdot b \cdot c(x)$ such that the trace $d(x) \cdot e(x)$ frees the resource x and is forbidden between $a(x)$ and $c(x)$.*

Let's define:

$$T_{a_1 \cdots a_n}(\mathcal{F}) = T_{Trace}(\mathcal{F}) \cdot a_1 \cdot T_{Trace}(\mathcal{F}) \cdots a_n \cdot T_{Trace}(\mathcal{F}).$$

We then define B by:

$$B = \bigcup_{\sigma} \left((a(x) \cdot T_b(\mathcal{F}) \cdot c(x))\sigma \setminus T_{(d(x) \cdot e(x))\sigma}(\mathcal{F}) \right).$$

Assume action b effectively realizes the behavior pattern functionality: abstraction with respect to this pattern then corresponds to inserting the abstraction symbol λ immediately after action b.

Traces realizing the behavior pattern are traces that verify one of the following conditions:

- *$d(x)$ appears between $a(x)$ and b, and $e(x)$ does not appear between $d(x)$ and $c(x)$;*
- *$e(x)$ appears between b and $c(x)$, and $d(x)$ does not appear between $a(x)$ and $e(x)$;*
- *$d(x)$ does not appear between $a(x)$ and b, and $e(x)$ does not appear between b and $c(x)$.*

Thus, we define three rewrite rules using the following $A_i(x)$ and $B_i(x)$ sets, respectively corresponding to the three above cases:

- *For the first condition, if $d(x)$ appears between $a(x)$ and b, then $e(x)$ does not appear between $d(x)$ and $c(x)$ iff $d(x) \cdot e(x)$ does not appear between $a(x)$ and b, and $e(x)$ does not appear between b and $c(x)$, which is expressed by:*
  - *$A_1(x) = \left(a(x) \cdot T_{d(x)}(\mathcal{F}) \cdot b\right) \setminus T_{d(x) \cdot e(x)}(\mathcal{F})$;*
  - *$B_1(x) = (T(\mathcal{F}) \cdot c(x)) \setminus T_{e(x)}(\mathcal{F})$.*
- *For the second condition, if $e(x)$ appears between b and $c(x)$, then $d(x)$ does not appear between $a(x)$ and $e(x)$ iff $d(x) \cdot e(x)$ does not appear between b and $c(x)$, and $d(x)$ does not appear between $a(x)$ and b, which is expressed by:*
  - *$A_2(x) = (a(x) \cdot T(\mathcal{F}) \cdot b) \setminus T_{d(x)}(\mathcal{F})$;*
  - *$B_2(x) = \left(T_{e(x)}(\mathcal{F}) \cdot c(x)\right) \setminus T_{d(x) \cdot e(x)}(\mathcal{F})$.*
- *The last condition is expressed by:*
  - *$A_3(x) = (a(x) \cdot T(\mathcal{F}) \cdot b) \setminus T_{d(x)}(\mathcal{F})$;*
  - *$B_3(x) = (T(\mathcal{F}) \cdot c(x)) \setminus T_{e(x)}(\mathcal{F})$.*

Importance of the choice of the insertion position for the abstraction symbol is illustrated by the following example.

Consider a behavior pattern describing the reading of a sensitive file $ReadFile$ and a behavior pattern $socket \cdot sendto$, describing the sending of data over the network. The trace $socket \cdot ReadFile \cdot sendto$ will be deemed suspicious only when the abstraction symbol λ_{read} identifying the reading of a sensitive file is inserted immediately after $ReadFile$ and the abstraction symbol λ_{send} identifying the sending of data over the network is inserted after $sendto$, yielding the abstracted trace $socket \cdot ReadFile \cdot \lambda_{read} \cdot sendto \cdot \lambda_{send}$. Indeed, in that case the trace will be interpreted as the the reading of a sensitive file followed by a network communication. If the abstraction symbol λ_{read} for $ReadFile$ had been inserted later in the trace, for example after $sendto$, we would have losed the sequence "read-send". The choice of this insertion position is therefore important for reducing false positives and false negatives in the detection algorithm.

C. Proofs

Theorems 1 and 2 rely on a lemma stating that, whenever some behavior pattern is abstracted within a trace t after any number of abstraction steps, it can be abstracted from t at the same concrete position and at the first abstraction step.

Definition 11. *Let t be a term of $T_{Trace}(\mathcal{F})$ and t' be a subterm of t, of sort $Trace$. The* concrete position of t' *in t is the position of $t'|_\Sigma$ in $t|_\Sigma$.*

Definition 12. *Let B be a behavior pattern associated with an abstraction symbol $\lambda \in \Gamma$ and with an abstraction relation $\rightarrow$. We say that the trace $t = t_1 \cdot t_2$ is* abstracted with respect to B into $t_1 \cdot \lambda \cdot t_2$ at the concrete position p, *denoted by $t_1 \cdot t_2 \rightarrow_p t_1 \cdot \lambda \cdot t_2$, iff $t_1 \cdot t_2 \rightarrow t_1 \cdot \lambda \cdot t_2$ and p is the concrete position of t_2 in t.*

The lemma can therefore be stated as follows. If $t \rightarrow^* t_1 \cdot t_2 \rightarrow_p t_1 \cdot \lambda \cdot t_2$, then there exists $u_1, u_2 \in T(\mathcal{F})$ such that: $t \rightarrow_p u_1 \cdot \lambda \cdot u_2$.

We actually show a more general form of this lemma, where a variable number of behavior patterns (not necessarily distinct) are abstracted one after the other.

Lemma 2. *Let $t \in T_{Trace}(\mathcal{F})$ be a trace and $\lambda_1, \lambda_2, \ldots \lambda_k \in T_{Action}(\mathcal{F}_\Gamma)$ be abstract actions. Let an abstraction chain from t be $t \rightarrow^* t_1 \cdot t'_1 \rightarrow_{p_1} t_1 \cdot \lambda_1 \cdot t'_1 \rightarrow^* t_2 \cdot t'_2 \rightarrow_{p_2} t_2 \cdot \lambda_2 \cdot t'_2 \rightarrow^* \ldots \rightarrow^* t_k \cdot t'_k \rightarrow_{p_k} t_k \cdot \lambda_k \cdot t'_k$ where we distinguish k abstraction steps, then:*

$$\exists u_1, \ldots u_k, u'_1, \ldots u'_k \in T_{Trace}(\mathcal{F}),$$
$$t \rightarrow_{p_1} u_1 \cdot \lambda_1 \cdot u'_1 \rightarrow_{p_2} u_2 \cdot \lambda_2 \cdot u'_2 \rightarrow_{p_3} \ldots \rightarrow_{p_k} u_k \cdot \lambda_k \cdot u'_k.$$

Proof: By induction on the length l of the derivation $t \rightarrow^* t_k . \cdot \lambda_k \cdot t'_k$.

- For the base case $l = 1$, we have: $t \rightarrow_{p_1} t_1 \cdot \lambda_1 \cdot t'_1$. Hence, there exists u_1, u'_1, $u_1 = t_1, u'_1 = t'_1$.

- For the general induction step, assume the property for $l = n$. We prove the property for $l = n+1$. By the induction hypothesis applied to $t \rightarrow^* t_1 \cdot t'_1 \rightarrow_{p_1} t_1 \cdot \lambda_1 \cdot t'_1 \rightarrow^* t_2 \cdot t'_2 \ldots \rightarrow^* t_k \cdot t'_k \rightarrow_{p_k} t_k \cdot \lambda_k \cdot t'_k$, $\exists u_1, \ldots u_k, u'_1, \ldots, u'_k \in T_{Trace}(\mathcal{F}), t \rightarrow_{p_1} u_1 \cdot \lambda_1 \cdot u'_1 \rightarrow \ldots \rightarrow u_k \cdot \lambda_k \cdot u'_k$.
  For $l = n+1$, the chain of length n is extended by $t_k \cdot \lambda_k \cdot t'_k \rightarrow t_{k+1} \cdot \lambda_{k+1} \cdot t'_{k+1}$.
  We want to rewrite $u_k \cdot \lambda_k \cdot u'_k$ into $u_{k+1} \cdot \lambda_{k+1} \cdot u'_{k+1}$. Now, existence of the reduction $t_k \cdot \lambda_k \cdot t'_k \rightarrow t_{k+1} \cdot \lambda_{k+1} \cdot t'_{k+1}$ entails the existence of an occurrence of the behavior pattern B_{k+1} in $t_k \cdot \lambda_k \cdot t'_k$. This occurrence also appears in $u_k \cdot \lambda_k \cdot u'_k$ and can therefore be abstracted at the same concrete position p_{k+1}, hence the existence of terms u_{k+1} and u'_{k+1} such that: $u_k \cdot \lambda_k \cdot u'_k \rightarrow_{p_{k+1}} u_{k+1} \cdot \lambda_{k+1} \cdot u'_{k+1}$. ■

Theorem 1. *Let Y be a set of variables of sort Data. Let $\alpha_1, \ldots, \alpha_m \in T_{Action}(\mathcal{F}_\Gamma, Y)$. Then the abstract behavior $M := \exists Y. \alpha_1 \odot \alpha_2 \odot \ldots \odot \alpha_m$ has the property of $(m, 0)$-completeness.*

Proof:
Let $L \subseteq T_{Trace}(\mathcal{F}_\Sigma)$ be a set of traces. We show that:

$$L \Subset M \Leftrightarrow \exists t' \in R^{\leq m}(L), \forall t'' \in R^{\leq 0}(t')\big|_\Gamma, t'' \models \varphi_M.$$

$\Rightarrow$: By Definition 3, there exists a trace $t \in L$ with a normal form $t\downarrow$ such that $t\downarrow|_\Gamma$ validates φ_M. Thus, $t\downarrow|_\Gamma$ can be written:

$$t\downarrow|_\Gamma = u \cdot v \cdot w$$

where $u, w \in T_{Trace}(\mathcal{F}_\Gamma)$ and $v \in M$. By definition of M, v is a trace validating the formula $\exists Y. \alpha_1 \odot \alpha_2 \odot \ldots \odot \alpha_m$ so, by the semantics of FOLTL, there exists a substitution σ_Y such that v validates the formula $\alpha_1\sigma_Y \odot \alpha_2\sigma_Y \odot \ldots \odot \alpha_m\sigma_Y$ which is equivalent to the formula $\alpha_1\sigma_Y \wedge \mathbf{X}(\top \mathbf{U} \alpha_2\sigma_Y \wedge \mathbf{X}(\top \mathbf{U} \ldots \wedge \mathbf{X}(\top \mathbf{U} \alpha_m\sigma_Y)))$. So v validates formulas $\alpha_1\sigma_Y$ and $\mathbf{X}(\top \mathbf{U} \alpha_2\sigma_Y \wedge \mathbf{X}(\top \mathbf{U} \ldots \wedge \mathbf{X}(\top \mathbf{U} \alpha_m\sigma_Y)))$. Therefore v is of the form:

$$v = \alpha_1\sigma_Y \cdot v_1 \cdot \alpha_2\sigma_Y \cdot v_2 \cdots \alpha_m\sigma_Y \cdot v_m$$

where $v_1, \ldots, v_m \in T_{Trace}(\mathcal{F}_\Gamma)$.

Since the projection of $t\downarrow$ removed actions from $T_{Action}(\mathcal{F}_\Sigma)$, $t\downarrow$ can be written:

$$t\downarrow = t_0 \cdot \alpha_1\sigma_Y \cdot t_1 \cdot \alpha_2\sigma_Y \cdot t_2 \cdots \alpha_m\sigma_Y \cdot t_m$$

where $t_0, \ldots, t_m \in T_{Trace}(\mathcal{F})$.

By Lemma 2, there exists $u_0, \ldots, u_m \in T_{Trace}(\mathcal{F})$ such that t is abstracted into $t' = u_0 \cdot \alpha_1\sigma_Y \cdot u_1 \cdots \alpha_m\sigma_Y \cdot u_m$ in exactly m steps. Thus $t' \in R^{\leq m}(L)$. Moreover, $R^{\leq 0}(t')\big|_\Gamma = \{t'|_\Gamma\}$ and $t'|_\Gamma \models \varphi_M$.

$\Leftarrow$: Let $t' \in R^{\leq m}(L)$ be a partial abstraction of a trace of L such that $\forall t'' \in R^{\leq 0}(t')\big|_\Gamma, t'' \models \varphi_M$. Then t' can be written $t' = t_0 \cdot \alpha_1\sigma_Y \cdot t_1 \cdots \alpha_m\sigma_Y \cdot t_m$, where $t_0, \ldots, t_m \in T_{Trace}(\mathcal{F})$ and $\sigma_Y \in Subst_Y$. Clearly, any future abstraction of t' will still be of the form $u_0 \cdot \alpha_1\sigma_Y \cdot u_1 \cdots \alpha_m\sigma_Y \cdot u_m$ and this will especially be true for any normal form $t'\downarrow$ of t' by R. Hence $t'\downarrow|_\Gamma \models \varphi_M$ and thus $L \Subset M$. ■

Theorem 2. *Let $M := \exists Y. \lambda_1(\overline{x_1}) \wedge \neg(\exists Z. \lambda_2(\overline{x_2}))\, \mathbf{U}\, \lambda_3(\overline{x_3})$ be an abstract behavior where Y and Z are two disjoint sets of variables of sort Data and where $\lambda_2 \neq \lambda_1$, $\lambda_2 \neq \lambda_3$ and λ_2 is independent from λ_3. Then M has the property of $(2, 1)$-completeness.*

Proof: Let's denote $\lambda_1(\overline{x_1})$, $\lambda_2(\overline{x_2})$ and $\lambda_3(\overline{x_3})$ by α_1, α_2 and α_3 respectively.

Let $L \subseteq T_{Trace}(\mathcal{F}_\Sigma)$ be a set of traces. We show that:

$$L \Subset M \Leftrightarrow \exists t' \in R^{\leq 2}(L), \forall t'' \in R^{\leq 1}(t')\big|_\Gamma, t'' \models \varphi_M.$$

$\Leftarrow$ We reason by contradiction. Let $t' \in R^{\leq 2}(L)$ be a trace such that $\forall t'' \in R^{\leq 1}(t')\big|_\Gamma, t'' \models \varphi_M$. Assuming that L does not exhibit M, we construct a trace $t'_1 \in R^{\leq 1}(t')$ which does not realize M and use the hypothesis $\forall t'' \in R^{\leq 1}(t')\big|_\Gamma, t'' \models \varphi_M$ to get a contradiction.

In particular, $t'|_\Gamma \models \varphi_M$ so, like in the proof of Theorem 1, by definition of the satisfiability of the formula $M := \exists Y. \lambda_1(\overline{x_1}) \wedge (\neg(\exists Z. \lambda_2(\overline{x_2}))\, \mathbf{U}\, \lambda_3(\overline{x_3}))$, there exists a substitution $\sigma_Y \in Subst_Y$ such that we can decompose t' into

$$t' = t_1 \cdot \alpha_1\sigma_Y \cdot t_2 \cdot \alpha_3\sigma_Y \cdot t_3$$

where $t_1, t_2, t_3 \in T_{Trace}(\mathcal{F})$, and such that there exists no substitution $\sigma_Z \in Subst_Z$ such that $\alpha_2\sigma_Y\sigma_Z$ appears in t_2.

Assume L does not exhibit M. Let t'' be a normal form of t': $t'' \in \{t'\}\downarrow_R$. Then, by Definition 3, $t''|_\Gamma \not\models \varphi_M$. By definition of the satisfiability of the formula $M := \exists Y. \lambda_1(\overline{x_1}) \wedge (\neg(\exists Z. \lambda_2(\overline{x_2}))\, \mathbf{U}\, \lambda_3(\overline{x_3}))$, there must exist a substitution $\sigma_Z \in Subst_Z$ such that the abstract action $\alpha_2\sigma_Y\sigma_Z$ has been inserted into a term of the derivation from t' to t'', at a concrete position p between $\alpha_1\sigma_Y$ and $\alpha_3\sigma_Y$. By Lemma 2, we could have inserted this action $\alpha_2\sigma_Y\sigma_Z$ directly in the term t', at the same concrete position p:

$$\exists u, w,\ t' \rightarrow_p u \cdot \alpha_2\sigma_Y\sigma_Z \cdot w.$$

Considering that the insertion is made between actions $\alpha_1\sigma_Y$ and $\alpha_3\sigma_Y$, and given that $t' = t_1 \cdot \alpha_1\sigma_Y \cdot t_2 \cdot \alpha_3\sigma_Y \cdot t_3$, we can decompose t_2 into $t_2 = t_2^1 \cdot t_2^2$ such that insertion occurs after t_2^1, in other words:

$$t' \rightarrow_p t_1 \cdot \alpha_1\sigma_Y \cdot t_2^1 \cdot \alpha_2\sigma_Y\sigma_Z \cdot t_2^2 \cdot \alpha_3\sigma_Y \cdot t_3.$$

Let's denote by t'_1 the obtained term: $t'_1 = t_1 \cdot \alpha_1\sigma_Y \cdot t_2^1 \cdot \alpha_2\sigma_Y\sigma_Z \cdot t_2^2 \cdot \alpha_3\sigma_Y \cdot t_3$. Then $t'_1 \in R^{\leq 1}(t')$ and yet $t'_1|_\Gamma \not\models \varphi_M$, which contradicts the hypothesis $\forall t'' \in R^{\leq 1}(t')\big|_\Gamma, t'' \models \varphi_M$.

$\Rightarrow$ By definition of the infection, there exists a trace $t \in L$ such that one of its normal forms $t\downarrow$ validates φ_M and can therefore be written:

$$t\downarrow = t_1 \cdot \alpha_1\sigma_Y \cdot t_2 \cdot \alpha_3\sigma_Y \cdot t_3$$

where $\sigma_Y \in Subst_Y$ is a ground substitution over Y, $t_1, t_2, t_3 \in T_{Trace}(\mathcal{F})$ and there exists no substitution σ_Z such that $\alpha_2\sigma_Y\sigma_Z$ appears in t_2.

We first define a term $t' \in R^{\leq 2}(t)$ that contains the same occurrence $\alpha_1\sigma_Y \cdot \alpha_3\sigma_Y$ of M and we show that its future abstractions remain infected. Indeed, as $t{\downarrow} = t_1 \cdot \alpha_1\sigma_Y \cdot t_2 \cdot \alpha_3\sigma_Y \cdot t_3$, by Lemma 2, there exists traces u', v', w' such that:

$$\exists u', v', w' \in T(\mathcal{F}_\Sigma),\ t \rightarrow\!\!\!\!\rightarrow u' \cdot \alpha_1\sigma_Y \cdot v' \cdot \alpha_3\sigma_Y \cdot w'.$$

Thus we define $t' = u' \cdot \alpha_1\sigma_Y \cdot v' \cdot \alpha_3\sigma_Y \cdot w'$. Moreover, since $t \in L$ is concrete, v' contains no abstract action, so it contains no instance of $\alpha_2\sigma_Y$. Hence, $t'|_\Gamma \models \varphi_M$.

We now show that: $\forall t'' \in R^{\leq 1}(t')|_\Gamma,\ t'' \models \varphi_M$. In fact, we show more generally that: $\forall t'' \in R^*(t')|_\Gamma,\ t'' \models \varphi_M$. Assume this is not the case and that t' can thus be rewritten in such a way that an action $\alpha_2\sigma_Y\sigma'_Z$ is inserted within v' for some substitution $\sigma'_Z \in Subst_Z$. The occurrence of the behavior pattern related to this insertion must also appear in $t{\downarrow}$. However, $t{\downarrow}$ is in normal form so this occurrence has been abstracted, at the same concrete position, in a term of the abstraction derivation from t to $t{\downarrow}$, that is after a concrete action of t_2.

Moreover, by hypothesis, in $t{\downarrow}$, no instance of $\alpha_2\sigma_X$ appears in t_2 since $t{\downarrow}$ validates φ_M.

So action $\alpha_2\sigma_Y\sigma'_Z$ would necessarily appear in the abstract actions at the head of t_3. But, since λ_2 and λ_3 are independent, this is impossible. Hence, no abstraction from t' could insert this action $\alpha_2\sigma_Y\sigma'_Z$ between $\alpha_1\sigma_Y$ and $\alpha_3\sigma_Y$.

Hence: $\forall t'' \in R^*(t')|_\Gamma,\ t'' \models \varphi_M$.

■

Theorem 3. *Let M be an abstract behavior defined by a formula $\varphi_M = \exists Y.\, \lambda_1(\overline{x_1}) \wedge \neg(\exists Z.\, \lambda_2(\overline{x_2}))\ \mathbf{U}\ \lambda_3(\overline{x_3})$ where Y and Z are disjoint sets of variables of sort Data and where $\lambda_2 \neq \lambda_1$, $\lambda_2 \neq \lambda_3$ and λ_2 is independent from λ_3. Then, for any set of traces $L \subseteq T_{Trace}(\mathcal{F}_\Sigma)$, L exhibits M iff:*

$$\exists t \in R_{\lambda_2}{\downarrow}(R^{\leq 2}(L))|_\Gamma,\ t \models \varphi_M.$$

Proof: $\Rightarrow$: By Theorem 2, M has the property of $(2,1)$-completeness so, by Definition 4, there exists a trace $t' \in R^{\leq 2}(L)$ such that $\forall t'' \in R^{\leq 1}(t')|_\Gamma,\ t'' \models \varphi_M$. In fact, we showed in the proof of Theorem 2 that: $\forall t'' \in R^*(t')|_\Gamma,\ t'' \models \varphi_M$. Since $R_{\lambda_2}{\downarrow} \subseteq R^*_{\lambda_2} \subseteq R^*$, we have:

$$\forall t'' \in R_{\lambda_2}{\downarrow}(t'),\ t''|_\Gamma \models \varphi_M.$$

Since the abstraction relation R is terminating, the set $R_{\lambda_2}{\downarrow}(t')$ is not empty, so this entails:

$$\exists t'' \in R_{\lambda_2}{\downarrow}(t'),\ t''|_\Gamma \models \varphi_M.$$

Considering that, by hypothesis, $t' \in R^{\leq 2}(L)$, we get:

$$\exists t \in R_{\lambda_2}{\downarrow}\left(R^{\leq 2}(L)\right)|_\Gamma,\ t \models \varphi_M.$$

$\Leftarrow$: Let $t'' \in R_{\lambda_2}{\downarrow}\left(R^{\leq 2}(L)\right)$ be a trace such that: $t''|_\Gamma \models \varphi_M$. Every occurrence of the behavior pattern λ_2 has been abstracted in t'' so any future abstraction of t'' by R only inserts abstract actions from $T_{Action}\left(\mathcal{F}_{\Gamma\setminus\{\lambda_2\}}\right)$, hence: $\forall u \in R^*(t'')|_\Gamma,\ u \models \varphi_M$. So any normal form of t'' by R is infected by M. Since R is terminating, t'' has at least one normal form with respect to R so, by Definition 3, L exhibits M.

■

Theorem 4. *Let R be an abstraction relation, such that R and R^{-1} are rational. There exists a detection procedure deciding whether L exhibits M, for any regular set of traces L and for any regular abstract behavior M having the property of (m,n)-completeness for some positive integers m and n.*

Proof: Let's define $M' = \pi_\Gamma^{-1}(M)$, where π_Γ^{-1} is the inverse of the projection on Γ. By definition of the abstract behavior M, (m,n)-completeness of M can be restated as:

$$L \Cap M \Leftrightarrow \exists t' \in R^{\leq m}(L),\ R^{\leq n}(t') \subseteq M'.$$

Let's show that the right member of this equivalence is decidable.

Observe first that, for any set $A \subseteq T_{Trace}(\mathcal{F})$, any term $t \in T_{Trace}(\mathcal{F})$ and any integer $i \in \mathbb{N}$, t can be rewritten by R into some term of A in i steps iff some term of A can be rewritten by R^{-1} into t in i steps:

$$R^i(t) \cap A \neq \emptyset \Leftrightarrow t \in \left(R^{-1}\right)^i(A) \tag{1}$$

Hence:

$$\begin{gathered}
R^{\leq n}(t') \subseteq M' \\
\Leftrightarrow \\
\neg\left(R^{\leq n}(t') \cap (T_{Trace}(\mathcal{F}) \setminus M') \neq \emptyset\right) \\
\Leftrightarrow \\
\neg\left(\bigvee_{0\leq i\leq n} \left(R^i(t') \cap (T_{Trace}(\mathcal{F}) \setminus M') \neq \emptyset\right)\right) \\
\Leftrightarrow \\
\text{by (1)} \\
\neg\left(\bigvee_{0\leq i\leq n} t' \in \left(R^{-1}\right)^i (T_{Trace}(\mathcal{F}) \setminus M')\right) \\
\Leftrightarrow \\
\neg\left(t' \in \left(R^{-1}\right)^{\leq n} (T_{Trace}(\mathcal{F}) \setminus M')\right)
\end{gathered}$$

Intuitively, this set $\left(R^{-1}\right)^{\leq n}(T_{Trace}(\mathcal{F}) \setminus M')$ represents the set of unreliably infected traces to avoid. Let's denote by M'' its complement: $M'' = T_{Trace}(\mathcal{F}) \setminus \left(R^{-1}\right)^{\leq n}(T_{Trace}(\mathcal{F}) \setminus M')$. So:

$$R^{\leq n}(t') \subseteq M' \Leftrightarrow t' \in M''.$$

The property of (m,n)-completeness can be restated as follows:

$$L \Cap M \Leftrightarrow R^{\leq m}(L) \cap M'' \neq \emptyset.$$

M is regular by hypothesis, so M' is regular too. Also, R^{-1} is rational so it preserves regularity, hence M'' is regular. Similarly, R is rational and L is regular, so $R^{\leq m}(L)$ is regular too, hence the decidability of detection. ■

We use the following lemma to prove Theorem 5.

Lemma 3. *Let R be a terminating abstraction relation. Let M be a regular malicious behavior with the property of (m,n)-completeness for some positive integers m and n.*

If R^{-1} is rational, then the set of traces n-reliably infected by M with respect to R is regular.

Proof: This set is precisely the set M'' defined in Theorem 4.

The set $T_{Trace}(\mathcal{F}) \setminus \pi_\Gamma^{-1}(M)$ is regular since inverse projection and complementation preserve regularity.

Sets $\left(R^{-1}\right)^i (T_{Trace}(\mathcal{F}) \setminus \pi_\Gamma^{-1}(M))$ are regular by rationality of R^{-1}, as is their union for $1 \leq i \leq n$, and the complement of their union. ∎

Theorem 5. *Let R be an abstraction relation such that R and R^{-1} are rational. Let τ be a tree transducer realizing R. Let M be a regular abstract behavior with the property of (m,n)-completeness and A_M be a tree automaton recognizing the set of traces n-reliably realizing M w.r.t. R. Deciding whether a regular set of traces L, recognized by a tree automaton A, exhibits M takes $O\left(|\tau|^{m\cdot(m+1)/2} \times |A| \times |A_M|\right)$ time and space.*

Proof: Let's denote by M'' the set of traces n-reliably infected by M with respect to R. The proof of Theorem 4 relied on the following result:

$$L \Subset M \Leftrightarrow R^{\leq m}(L) \cap M'' = \emptyset$$

By Theorem 6, there is a tree automaton recognizing $R^{\leq m}(L)$ of size $O\left(|\tau|^{m\cdot(m+1)/2} \times |A|\right)$. Intersection of two tree automata A_1 and A_2 yields an automaton of size $O(|A_1| \times |A_2|)$. Finally, deciding whether an automaton recognizes the empty set takes time and space linear in its size. ∎

We use the following definition and lemma to prove the rationality of abstraction.

Definition 13. *Let Ω be a set of function symbols with profile $Data^n \to Action$, $n \in \mathbb{N}$. Let $L \subseteq T_{Trace}(\mathcal{F})$ be a set of traces. The Ω-generalized form of L, denoted by $\amalg_\Omega(L)$, is the set:*

$$\amalg_\Omega(L) = \{ \quad t_0 \cdot a_1 \cdot t_1 \cdots a_n \cdot t_n \mid \\ a_1, \ldots, a_n \in T_{Action}(\Omega \cup \mathcal{F}_d), \\ t_0 \cdots t_n \in L\}.$$

Lemma 4. *Let $\Omega \subseteq \mathcal{F}_a$ be a set of function symbols with profile $Data^n \to Action$, $n \in \mathbb{N}$. If A is a tree automaton, then there exists a tree automaton of size $O(|A|)$ recognizing the Ω-generalized form of $\mathcal{L}(A)$.*

Proof:

We define a top-down tree transducer $\tau = (\mathcal{F}, \mathcal{F} \cup \Omega, \{q_t, q_a, q_d\}, q_t, \Delta)$ such that: $R_\tau(\mathcal{L}(A)) = \amalg_\Omega(\mathcal{L}(A))$.

Δ is composed of the following rules:

- $q_t(\cdot(x_1, x_2)) \to \cdot(q_a(x_1), q_t(x_2))$;
- $q_t(\epsilon) \to \epsilon$;
- For all $k \in \mathbb{N}$, for all $f \in \mathcal{F}_a^{(k)}$, Δ contains a rule: $q_a(f(x_1, \ldots, x_k)) \to f(q_d(x_1), \ldots, q_d(x_k))$;
- For all $d \in \mathcal{F}_d$, Δ contains a rule: $q_d(d) \to d$;
- For all term $\alpha \in T_{Action}(\mathcal{F}_\Omega)$, Δ contains a rule: $q_t(x) \to \cdot(\alpha, q_t(x))$.

The transducer τ realizes $\amalg_\Omega$ and has a size constant with respect to the size of A. Lemma 1 thus entails that $\amalg_\Omega(\mathcal{L}(A))$ is recognized by a tree automaton of size $O\left(sz^r_{\mathcal{F}_{\Omega\cup\Omega'}}(\tau) \times |A| \times |\tau|\right) = O(|A|)$. ∎

Theorem 6. *Let B be a behavior pattern and R be a terminating abstraction relation w.r.t. B defined by an abstraction system whose set of instances of right-hand sides of rules is recognized by a tree automaton A_R. Then R and R^{-1} are rational and, for any tree automaton A, $R(\mathcal{L}(A))$ is recognized by a tree automaton of size $O(|A| \cdot |A_R|)$*

Proof: We construct two tree transducers realizing R and R^{-1}.

Let n be the number of rules of the abstraction system. Let C denote the set $\bigcup_{i\in[1..n]} \bigcup_{\sigma\in Subst_X} A_i(X)\sigma \cdot \lambda(\overline{x})\sigma \cdot B_i(X)\sigma \subseteq T_{Trace}(\mathcal{F})$.

Let $\diamondsuit$ be a constant of sort $Action$ not in $\mathcal{F}_a$. We consider the following set:

$$Img_\diamondsuit(R) = \{t_1 \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot t_2 \mid \quad t_1, t_2 \in T_{Trace}(\mathcal{F}), \\ \alpha \in T_{Action}(\mathcal{F}_\Gamma), \\ (t_1 \cdot t_2, t_1 \cdot \alpha \cdot t_2) \in R\}.$$

We will show that this set is recognized by a tree automaton, of size $O(|A_R|)$, and then use this set to construct a tree transducer recognizing R and R^{-1}.

Define C' to be the set:

$$C' = \amalg_{\{\diamondsuit\}}(C) \\ \cap \\ (T_{Trace}(\mathcal{F}) \cdot \diamondsuit \cdot T_{Action}(\mathcal{F}_\Gamma) \cdot \diamondsuit \cdot T_{Trace}(\mathcal{F})).$$

By Lemma 4 applied to $\Omega = \{\diamondsuit\}$ and to automaton A_R, the set $\amalg_{\{\diamondsuit\}}(C)$ is regular and recognized by a tree automaton, of size $O(|A_R|)$. The set $T_{Trace}(\mathcal{F}) \cdot \diamondsuit \cdot T_{Action}(\mathcal{F}_\Gamma) \cdot \diamondsuit \cdot T_{Trace}(\mathcal{F})$ is also recognized by a tree automaton of constant size. So their intersection C' is regular and recognized by a tree automaton $A_{C'}$ of size $O(|A_R|)$. Terms in C' are terms of C where the abstract action has been enclosed between two diamond symbols.

Now, define C'' to be the set

$$C'' = \amalg_\Gamma(C') \\ \cap \\ (T_{Trace}(\mathcal{F}) \cdot T_{Action}(\mathcal{F}_\Sigma) \cdot \\ \diamondsuit \cdot T_{Action}(\mathcal{F}_\Gamma) \cdot \diamondsuit \cdot T_{Trace}(\mathcal{F})).$$

By Lemma 4 applied to $\Omega = \Gamma$ and to automaton $A_{C'}$, the set $\amalg_\Gamma(C')$ is regular and recognized by a tree automaton of size $O(|A_R|)$. The set $T_{Trace}(\mathcal{F}) \cdot T_{Action}(\mathcal{F}_\Sigma) \cdot \diamondsuit \cdot T_{Action}(\mathcal{F}_\Gamma) \cdot \diamondsuit \cdot T_{Trace}(\mathcal{F})$ is also recognized by a tree automaton and of constant size. So their intersection C'' is regular and recognized by a tree automaton $A_{C''}$ of size $O(|A_R|)$.

Finally, we remove from C'' the terms violating the terminating condition of Definition 10. We define C''' to be the set:

$$C''' = C'' \setminus \bigcup_{\alpha \in T_{Action}(\mathcal{F}_\Gamma)} (T_{Trace}(\mathcal{F}) \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot T_{Trace}(\mathcal{F}_\Gamma) \cdot \alpha \cdot T_{Trace}(\mathcal{F})).$$

The set $T_{Action}(\mathcal{F}_\Gamma)$ is finite, so the set $\bigcup_{\alpha \in T_{Action}(\mathcal{F}_\Gamma)} T_{Trace}(\mathcal{F}) \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot T_{Trace}(\mathcal{F}_\Gamma) \cdot \alpha \cdot T_{Trace}(\mathcal{F})$ is recognized by a tree automaton of constant size. Hence, C''' is regular, recognized by a tree automaton of size $O(|A_{C''}|) = O(|A_R|)$, and it verifies:

$$T_{Trace}(\mathcal{F}) \cdot C''' \cdot T_{Trace}(\mathcal{F}) = Img_\diamondsuit(R). \quad (2)$$

Indeed, for all $t_1, t_2 \in T_{Trace}(\mathcal{F})$ and for all $\alpha \in T_{Action}(\mathcal{F}_\Gamma)$:

$$\begin{gathered}
(t_1 \cdot t_2, t_1 \cdot \alpha \cdot t_2) \in R \\
\Leftrightarrow \\
\exists u_1, v_1, u_2, v_2 \in T_{Trace}(\mathcal{F}), \\
t_1 = u_1 \cdot v_1, t_2 = v_2 \cdot u_2, \\
v_1|_\Sigma \cdot \alpha \cdot v_2|_\Sigma \in C, \\
v_1 \in T_{Trace}(\mathcal{F}) \cdot T_{Action}(\mathcal{F}_\Sigma), \\
v_2 \notin T_{Trace}(\mathcal{F}_\Gamma) \cdot \alpha \cdot T_{Trace}(\mathcal{F}) \\
\Leftrightarrow \\
\exists u_1, v_1, u_2, v_2 \in T_{Trace}(\mathcal{F}), \\
t_1 = u_1 \cdot v_1, t_2 = v_2 \cdot u_2, \\
v_1|_\Sigma \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot v_2|_\Sigma \in C', \\
v_1 \in T_{Trace}(\mathcal{F}) \cdot T_{Action}(\mathcal{F}_\Sigma), \\
v_2 \notin T_{Trace}(\mathcal{F}_\Gamma) \cdot \alpha \cdot T_{Trace}(\mathcal{F}) \\
\Leftrightarrow \\
\exists u_1, v_1, u_2, v_2 \in T_{Trace}(\mathcal{F}), \\
t_1 = u_1 \cdot v_1, t_2 = v_2 \cdot u_2, \\
v_1 \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot v_2 \in C'', \\
v_2 \notin T_{Trace}(\mathcal{F}_\Gamma) \cdot \alpha \cdot T_{Trace}(\mathcal{F}) \\
\Leftrightarrow \\
\exists u_1, v_1, u_2, v_2 \in T_{Trace}(\mathcal{F}), \\
t_1 = u_1 \cdot v_1, t_2 = v_2 \cdot u_2, \\
v_1 \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot v_2 \in C''' \\
\Leftrightarrow \\
t_1 \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot t_2 \in T_{Trace}(\mathcal{F}) \cdot C''' \cdot T_{Trace}(\mathcal{F}).
\end{gathered}$$

We now define the transducers realizing R and R^{-1}. To that end, let's consider the relation T defined by:

$$T = \{ \quad (t \cdot t', t \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot t'), \mid t, t' \in T_{Trace}(\mathcal{F}), \alpha \in T_{Action}(\mathcal{F}_\Gamma) \}.$$

Clearly, relations T and T^{-1} are rational and recognized by transducers τ_T and $\tau_{T^{-1}}$ of constant size.

The set $T_{Trace}(\mathcal{F}) \cdot C''' \cdot T_{Trace}(\mathcal{F})$ is recognized by a tree automaton of size $O(|A_R|)$, so there exists a tree transducer $\tau_{C'''}$ realizing the relation $\{(t,t) \mid t \in T_{Trace}(\mathcal{F}) \cdot C''' \cdot T_{Trace}(\mathcal{F})\}$ and of size $O(|A_R|)$.

Moreover, let $\tau_\diamondsuit$ be the tree transducer on $T_{Trace}(\mathcal{F} \cup \{\diamondsuit\})$ realizing the projection on $\Sigma \cup \Gamma$, i.e. removing the diamond symbol, and $\tau_{\diamondsuit^{-1}}$ be the tree transducer on $T_{Trace}(\mathcal{F} \cup \{\diamondsuit\})$ inserting random diamonds on the ouput. $\tau_\diamondsuit$ and $\tau_{\diamondsuit^{-1}}$ are of constant size.

Then R is realized by the tree transducer $\tau_\diamondsuit \circ \tau_{C'''} \circ \tau_T$. Indeed, for all $t_1, t_2 \in T_{Trace}(\mathcal{F}), \alpha \in T_{Action}(\mathcal{F}_\Gamma)$, let $t' = t_1 \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot t_2$, then:

$$\begin{gathered}
(t_1 \cdot t_2, t_1 \cdot \alpha \cdot t_2) \in R \\
\Leftrightarrow \\
t' = t_1 \cdot \diamondsuit \cdot \alpha \cdot \diamondsuit \cdot t_2 \in Img_\diamondsuit(R) \\
\Leftrightarrow \\
\text{by (2)} \\
t' \in T_{Trace}(\mathcal{F}) \cdot C''' \cdot T_{Trace}(\mathcal{F}) \\
\Leftrightarrow \\
(t_1 \cdot t_2, t') \in T, (t', t') \in R_{\tau_{C'''}} \text{ and } (t', t_1 \cdot \alpha \cdot t_2) \in R_{\tau_\diamondsuit} \\
\Leftrightarrow \\
(t_1 \cdot t_2, t_1 \cdot \alpha \cdot t_2) \in R_{\tau_\diamondsuit \circ \tau_{C'''} \circ \tau_T}.
\end{gathered}$$

Similarly, R^{-1} is realized by the tree transducer $\tau_{T^{-1}} \circ \tau_{C'''} \circ \tau_{\diamondsuit^{-1}}$. Indeed: $\left(R_{\tau_\diamondsuit}\right)^{-1} = R_{\tau_{\diamondsuit^{-1}}}$ and $\left(R_{\tau_{C'''}}\right)^{-1} = R_{\tau_{C'''}}$, hence:

$$R^{-1} = R_{\tau_{T^{-1}} \circ \tau_{C'''} \circ \tau_{\diamondsuit^{-1}}}.$$

Finally, for any tree automaton A, the set $R(\mathcal{L}(A)) = \tau_\diamondsuit(\tau_{C'''}(\tau_T(\mathcal{L}(A))))$ is recognized by a tree automaton of size $O(|A| \cdot |A_R|)$, by Lemma 1. ■